2010 Latin American Robotics Symposium and Intelligent Robotics Meeting

HW/SW Co-design Architecture for Evolutionary Robotics

Mauricio A Dias, Daniel O Sales and Fernando S Osorio
LRM, ICMC-USP, Sao Carlos - SP, Brazil
macdias, dsales, fosorio @ icmc.usp.br

***Abstract*— Evolutionary algorithms are very common techniques used in computational intelligence and robotics field applications. Some algorithms need a large amount of memory and processing power, making them difficult to implement into embedded systems. In this work a profile-based approach is proposed and applied in an evolutionary algorithm with some characteristics that allow it's use on embedded systems and robotics: the μGA. The main goal is to implement a new hardware-software co-design architecture for this genetic algorithm with better execution time than algorithms implemented in software (using general purpose hardware solutions). The presented results show a comparison between different co-design implementations and discussion about new architecture advantages.**

I. Introduction

In recent years some computational intelligence methods gained researcher's attention, specially Artificial Neural Networks and Evolutionary Algorithms [1][2]. One specific implementation of evolutionary algorithms is the Genetic Algorithm (GA), proposed in 70's by Holland and his students, and largely diffused in 1989 by Goldberg [1]. In this algorithm, the descendants are generated by operations between individuals of the population, and one of the most important is the crossover operator.

The algorithm executes two basic steps: create an initial population and begin the main loop that evolves this population. The main loop consists of making interactions with individuals (with crossover and mutation operators) and evaluating them, evolving/selecting the best individuals according to a fitness function. This loop is repeated until reach the stopping criteria, which may be a maximum number of generations, or finding an individual that achieve the desired fitness value. With this procedure, the algorithm is able to keep the best solutions and explore the search space at the same time.

Evolutionary algorithms have been presenting good results in many computational fields. Robotics is one of them. Evolutionary robotics is an area that is receiving increasing attention from robotics researchers last years. There are many possible applications for this kind of algorithms in robotics from robot's physical configuration to control systems and navigation [2].

Control systems for autonomous robots are often programmed by researchers or designers. As the complexity of the environment and tasks for autonomous robots increases, the difficulty of designing control systems by hand becomes a limiting factor, considering the degree of functional complexity that should be achieved [2]. One possible solution for this problem is to use automatic learning methods as evolutionary computing.

Robots can be seen as embedded systems or systems composed by a group of embedded systems. In most cases embedded systems have limited capacity of processing and memory. Develop embedded systems for robotics is a complex task because these systems share resources with sensors and actuators. The combination of these facts justifies the adoption of more robust techniques for embedded robotic system's design considering performance, costs, energy consumption, processing and execution time.

The first and most intuitive solution for embedded systems design is to implement algorithms directly in hardware. Hardware implementation projects nowadays have been replaced by hardware/software co-designs mainly on embedded systems design. Hardware/software co-design is a design method that proposes hardware and software concurrent development. Decide which part of the system will be implemented in hardware and which will be implemented in software is a classic problem on co-design called hardware/software partitioning.

This choice for hardware/software co-design is based on increasing complexity of embedded systems, reduction of time-to-market for embedded systems design and increased availability of hardware due to lower costs.

Co-design methods usually require flexible development tools that allows rapid prototyping. One way to achieve desired flexibility level is to develop hardware with reconfigurable computing devices such as FPGA's.

Reconfigurable computing can be defined as the study of computing with reconfigurable devices [3]. This paradigm tends to achieve high performance with high flexibility. To implement reconfigurable computing is necessary to use reconfigurable devices. Reconfigurable devices are devices that allow the process of changing its structure at run-time. There are various types of reconfigurable devices and one of them is specially interesting for this work, the FPGA's.

Field Programmable Gate Arrays (FPGA's) are programmable devices consisting of three main parts: programmable logic cells, configurable logic blocks and I/O cells [3]. FPGA's are flexible devices because, among various features, allow hardware to be described using Hardware Description Languages (HDL's) and several reconfigurations. In this work specifically, there is a free soft-processor provided by FPGA's manufacturer that can be used for co-design development.

A soft processor is an Intellectual Property (IP) core which is 100% implemented using the logic primitives of the FPGA.

978-0-7695-4231-7/10 $26.00 © 2010 IEEE
DOI 10.1109/LARS.2010.30

Other definition: a programmable instruction processor implemented in the reconfigurable logic of the FPGA [4]. Soft processors have several advantages and one of the most relevant for actual designs is the possibility to implement the exact number of soft-processors required by the application. Since it is implemented in configurable logic, a soft processor can be tuned by varying its implementation and complexity to match the exact requirements of an application [4].

This work proposes to implement, on a reconfigurable hardware (FPGA), an optimized Genetic Algorithm (μGA) that should be able to respect ”soft” real-time embedded systems requirements. To achieve system requirements, a profile-based hardware/software co-design method was selected and after system profiling analysis most critical functions were implemented in hardware. The proposed architecture for the chosen algorithm consists on a Nios II processor with custom instruction crossover hardware and a floating point unit (FPU) hardware for fitness optimization.

Section II presents some recent hardware development for GA works. In section III, concepts of μGA are presented followed by hardware/software co-design description. Section IV contains proposed method description followed by the results section. In results section (V), proposed hardware development is presented jointly with a comparison between different co-design implementations and discussion about new architecture advantages. At last section (VI) presents authors' conclusions based on achieved results followed by selected references.

II. Related Works

Evolutive algorithms are widely applied to solve problems in many fields including robotics. Due to this fact, hardware researchers have been working on efficient implementations for these algorithms. Large amounts of solutions were presented in the last years but some problems are common for these solutions and exemplified on following analysis.

Zhu et. Al. [5] proposed a 100% hardware implementation of a genetic algorithm to optimize memory requirement and access speed. The algorithm is partitioned on a global search and a local search. The problem in this case is that the results presented are from a MATLAB simulation, so there is no physical hardware implementation.

Fitness functions are usually the main problem of genetic algorithms, even in case of hardware implementations, since they are hard to implement on hardware once these functions can vary from application to application. Nedjah and Mourelle[6] proposed another hardware implementation for GA where fitness function was implemented as an artificial neural network. The work has a lack of details about hardware implementation of the neural network and no real implementation results of the GA.

Recent works present some interesting hardware architectures. Chen et. Al.[7] presented an IP (intellectual property) for GAs. The idea is to provide a flexible hardware implementation with four types of crossover (with tournament selection), a good range of population possible sizes, mutation rates, individual and fitness value bit length. Fitness function in this case can be implemented in two ways: a look-up table or an user defined circuit. The look-up table option is limited to the proposed IP structure so the flexibility is sacrificed in this case. Developing hardware implementations for fitness functions is usually a non-trivial task so the second option has problems either.

Oliveira and Junior[8] proposed a complete hardware implementation for a compact genetic algorithm that is only capable of solving first order problems. Furthermore, individual evaluation was not implemented in hardware. Kher et. Al.[9] developed a dynamic crossover hardware, that means, crossover dynamically changes the number of cut-points during execution. On presented results, the problem of dynamic crossover is that the algorithm should stop once the expected value is achieved and stabilized but the result starts to toggle between the achieved value and a lower bound. This problem is caused by the crossover.

The common problems of this section's presented works are: (i) the implementations are not completely self-contained embedded systems, that means, hardware needs information coming from another hardware structure [7] or from a general-purpose computer [9] to work; (ii) there is no physical implementation [5][6]; or (iii) the algorithm cannot solve complex problems [8].

The architecture proposed here were developed in order to be an alternative to previous works and solve the main problems.

III. μ Genetic Algortithm

Micro Genetic Algorithm (μGA) is a genetic algorithm with very small population and simple genetic parameters, used for solving function optimization problems. It was proposed by Krishnakumar in 1989 [10] as a faster alternative to Simple GA [1] and other usual implementations of GA's.

The result achieved in Krishnakumar's work with non-linear functions shows that its implementation is quicker than big population approaches in spite of its simplicity and achieves the same results. As the number of individuals is small, it is possible to evaluate the fitness function and perform the genetic operators more rapidly.

It is known that small populations converge to non-optimal results sometimes, due to insufficient information processing and low diversity between individuals. The solution for this problem is: (i) to transfer the best individual to a new population using elitism; (ii) to select the best individuals with a higher probability for crossover; and (iii) to generate from time-to-time a set of new individuals randomly avoiding early convergence. In (iii) approach, a small size population is generated randomly, and genetic operations are performed until reach the nominal convergence. Then, the best individual is transferred to a new population and the remaining individuals are generated randomly and after that the algorithm goes back to the second step and repeat until reach the stopping criteria.

Unlike in other GA implementations, the solution can be found by evaluating the fitness of the best individual, not only when all individuals converge to a single value. This is also an advantage which makes μGA end faster. The crossover rate is usually 1, and mutation rate must be near to 0, because enough diversity is introduced every time the population is re-initialized (epidemic operator).

Micro-Genetic Algorithms are useful in many applications, for example Real-Time Systems, and particularly in Evolutionary Robotics. In some applications each robot can be represented as one individual instead of having many robots used to optimize/find the problem solution, which can usually be unfeasible.

IV. Hardware/Software Co-Design

Electronic systems nowadays have become more complex because of the rising number of functionalities that are requested on projects. As electronic circuits become more complex, technical challenges in co-design will also increase [11]. These circuit systems can be classified as a joined operation component group developed for some task resolution [12] and, this components, can be implemented in hardware or in software depending on what are the system's constraints.

Hardware/Software co-design means the union of all system goals exploring interaction between hardware and software components during development [13]. Fig. 1 presents a simplified co-design flow. This method tries to increase the predictability of embedded system design by providing analysis methods that tell designers if a system meets its performance, power, size and synthesis methods that let researchers and designers rapidly evaluate many potential design methodologies [11].

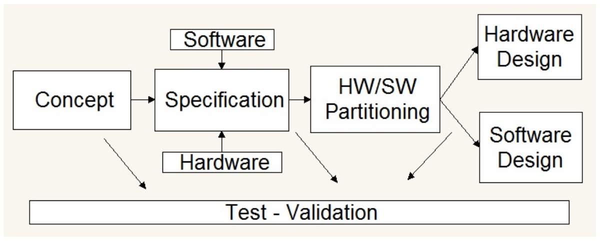

Fig. 1. Co-design Flow.

Hardware/Software co-design is a hard task and computational tools appeared to make this work faster and increases system's degree of optimization. The introduction of programmable devices like FPGA's (Field Programmable Gate Arrays) on circuit co-design enabled flexibility and made prototyping easier. This fact is really important because the circuit can be developed and tested before be manufactured, reducing costs and design time. This flexibility opens new digital circuit applications and the rise of new Hardware/Software co-design problems [14].

Hardware/software partitioning problem has a first level impact on system's performance because a wrong choice can result in an increased development time and a final product out of specification. A good partitioning that fit all implementation requirements optimizes operation and minimizes costs usually resulting on an efficient project.

First, partitioning problem was solved by developers with their experience and previously released works. In case of complex systems, this work became arduous and this fact inducted this research field interest on automating the process using techniques like evolutionary computing and artificial neural networks [15]. This automation can be achieved with EDA (electronic design automation) tools or methods that allow a fast search for good solutions on search space. Among existing methods, the profiling-based methods are being widely used nowadays.

A. *Profiling-Based Methods*

There are many methods for hardware/software co-design and the most commonly used methods nowadays are the profiling-based methods [16].

TABLE I
Profiling Tool's Comparison.

	Memory	Power	Per Funct	Per Line	Call Grph
Gprof	-	-	x	x	x
HDL Profiling	x	x	-	-	-
ATOMIUM	x	-	x	-	x
MEMTRACE	x	x	x	x	-

The increasing complexity of codes raised the need for profiling tools. Profiling tools make code analysis and indicate several system features like functions execution time and memory usage. Each tool gives a specific group of information and most of them have clock cycle information and table I presents some other relevant information [16].

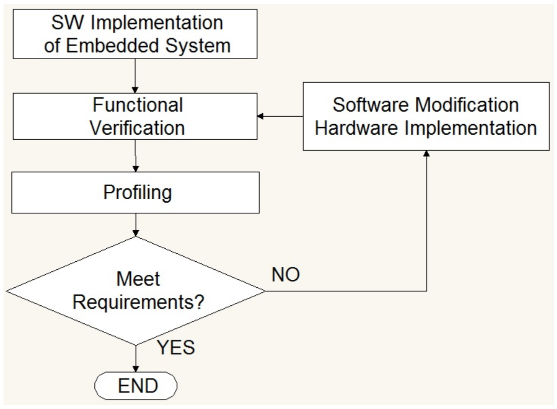
SW Implementation of Embedded System
Functional Verification
Software Modification Hardware Implementation
Profiling
Meet Requirements?
NO
YES
END

Fig. 2. General Profile-Based Method [17].

After obtaining profiling information, system can be modified to achieve expected performance. Usually the critical parts of the system (intensive processing and time consuming routines) start been optimized with code modification and improvement, followed by hardware development on extreme cases. This is a cyclic refinement process and usually stops when co-design performance constraints or maximum time-to-market are achieved. Fig. 2 illustrates general profile-based method design flow.

One of the most important features of this kind of methods is that the critical part of the co-design, hardware/software partitioning, is performed during a practical refinement process. Due to this fact there is a high probability that the final system has one of the best partitioning contained in the search space.

Nowadays hardware/software co-design is using soft-processors because they allow software execution and designed hardware to be included as custom instructions for validation.

V. Method Implementation

Previous sections presented the context that this work is inserted. Considering these concepts the main idea is to implement μGA in an embedded platform and achieve an acceptable execution time for "soft" real time robotic applications. First the co-design method will be applied in order to select the configuration of the soft-processor that fits better system constraints as area consumption, execution time, development time. After that hardware components will be developed according to profile results and execution time will be compared.

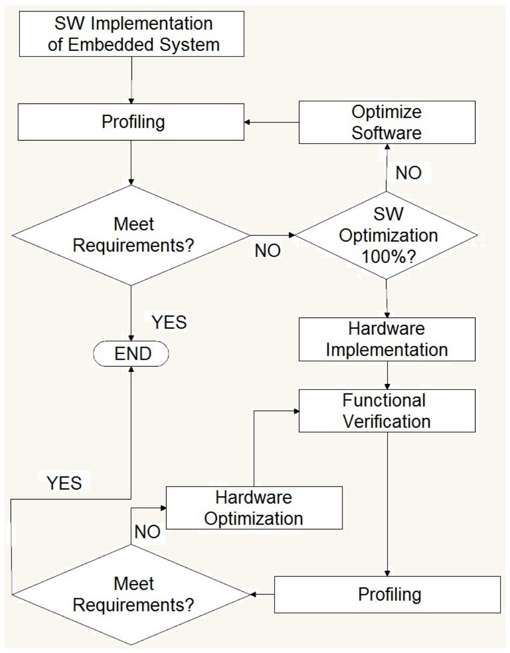

Fig. 3. Modified Profile-Based Method.

Fig. 3 illustrates flowchart of proposed hardware/software co-design method. There are two main cycles, the first one of software development and the second one of hardware development. Sometimes system requirements can be satisfied only doing software optimizations. When software optimizations reach the limit without satisfying requirements, hardware development starts. On embedded systems design hardware development is a costly task comparing to software development, so a large amount of time can be saved choosing this modified method. Together with this fact the final solution can be more interesting in many characteristics like cost, performance and energy because only the portion of the system that needs acceleration will be implemented in hardware.

To evaluate proposed hardware some fitness function had to be chosen. μGA is an algorithm with interesting characteristics for robotics due to it's small population and a robotic fitness function to evaluate this algorithm seems to be an interesting choice. Any fitness function would serve the purpose and a reactive LIDAR-based function for reactive control was chosen.

The chromosome is coded with six bits (b_0-b_5), one for left, one for left-center, two for center, one for center-right and one for right. The chromosome gives the robot direction. This codification is easily transformed on robot actuator instructions when coded on Player/Stage environment [18], mapping each state into a set of actuator function parameters. LIDAR range is divide into six quadrants (q_0-q_5) selecting the minor measured value as quadrant correspondent value. Fig. 4 illustrates these considerations.

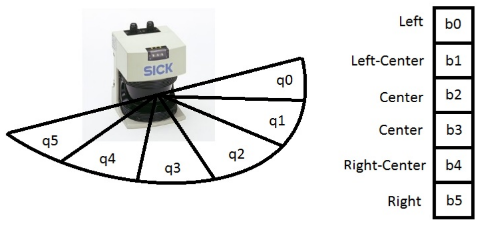

Fig. 4. Genetic Codification.

Fitness function, that is a maximization function presented by equation (1), calculated finding the max value of the multiplications b_n * q_n and subtracting other values form it. The idea of this algorithm is to control the navigation of the robot avoiding obstacles, so it needs to run in real-time while laser values change. The algorithm follows a basic cycle of reading sensor values, execute μGA algorithm and send actuator function parameters. It's a simple fitness functions but the main goal of this work is to define and validate a hardware architecture for μGA not to propose innovative fitness functions, or to find other genetic programming based algorithms to use for many reasons explained on previous sections.

$$f_{individual} = f_{max}(b_n * q_n | n \leq 5) - \sum_{0}^{5} (b_n * q_n | n \leq 5, \neq n_{max}) \quad (1)$$

In order to accelerate crossover function, a new way to do selection is proposed. Krishnakumar [10] suggested crossover rate on 100%, mutation rate on 0%, epidemic operator and elitism. There are many selection methods, but finding possible individuals to cross is always a hard task with many verifications. The idea of this work's proposed crossover (figure 5) is to create a new vector filled with the number of individuals (0,1,2,3..) and shuffling this vector, respecting elitism (the best individual on the first position of population vector and never substituted). In the end of this operation the vector has the pairs for crossover ready. Crossover results on two new individuals composed by, after choosing the cut-point (floor((#bits in chromosome)/2), first part of the first individual followed by second part of second individual and second part of first individual followed by first part of second individual. For odd number of individuals the last part is taken from the best, it happened to this work since the number of individuals is five. The number of generations was set to 50. This static crossover keeps the size of the population, applies the crossover for all individuals, keep the best individual. The shuffle function is faster than other selection methods because time is not wasted on testing and selecting individuals. Local minima are avoided by the fact that not only the higher fitness function individuals do crossover, together with epidemic operator usage.

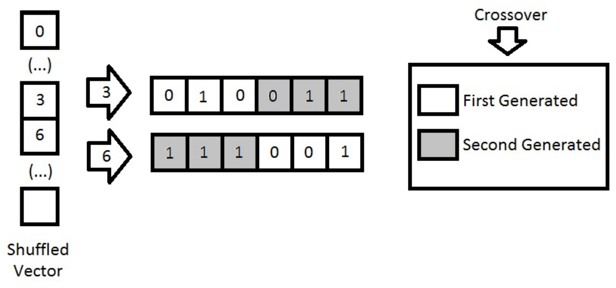

Fig. 5. Crossover example.

After using the proposed crossover, hardware crossover implementation became easier. The shuffle vector, described in section IV, stored population vector with the new order of individuals. Chosen soft-processor has custom instructions templates with two inputs of 32 bits and one output of 32 bits, and only these I/Os are needed in this case. A hardware for crossover was build based on this specific implementation's crossover, but on a way that is easy to think about higher number of individual bits and higher number of individuals. Custom instruction implementation is better for this hardware because there's no communications costs since it will be considered an extension of Nios II ALU unit. For more information about how custom instructions are implemented in Nios II see the Altera documentation [19].

It's a simple hardware implemented on Nios II /ff because of the better execution time and acceptable area consumed. Reordered vector is the input of the hardware with two extra positions to complete 32 bits and the output is another 32-bit vector after crossover. Inside this hardware the signals only take their new places based on one cut-point crossover. This operation takes only one clock cycle and, in this case, five individuals only need onde hardware call.

There are many features that can be evaluated on a hardware/software co-design. In this case the most important is execution time, but together with execution time hardware developments should achieve good values on: hardware area; maximum system clock for critical path; presence of pipelined or not microprocessor; dedicated modules for specific operations and energy consumption. Development time is also important on co-design specially when associated with time-to-market.

Several available development environments and toolboxes could be used in this work's experiments. Due to researches experience, experiments were realized on Altera Quartus II IDE + Nios II EDS and implemented on DE2-70's Cyclone II family FPGA manufactured by Altera. This family is composed by hierarchy-based FPGA's, that means, there are basic logic blocks that are grouped to form larger blocks and so on. Blocks are connected by programmable interconnections and the chip is rounded by input/output pins. Nios II EDS supports Altera's soft processor Nios II that can be configured on Quartus II SOPC Builder. IDE's, information and manuals can be downloaded in [19]. Nios II soft-processor has three types of implementations with different degrees of complexity and supports hardware to be integrated as processor's custom instructions. So developed hardware was included as custom instruction for evaluation. Analyzed profiles were obtained with GProf (Gnu Profiler) [20] that is integrated with Nios II EDS, that has also a gcc compiler for Nios II processors so software development was made on C language.

VI. Results

To evaluate the influence of soft-processor feature changing and achieve good results on co-design methodology, the experiments have been done with ten different types of hardware (soft-processor + custom instructions) and each one executing the μGA described in previous sections. Before doing any hardware development some code modifications have been done. Basically parameters were adjusted on pc coded algorithms to reach the minimum number of generations that return the expected results with precision of two decimal places. Before executing on the soft-processor all benchmark function solutions were programmed to execute instantaneously on a dual-core Intel Pentium's.

Starting with pipelined processor influence on execution time and area consumption, three processors have been configured for these experiments initially: Nios II economic (e), fast (f) and standard (s). Soft-processors had the same basic configuration with the CPU, 100K of on-chip memory, timer for clock and JTAG interface for communication. The main difference between these three processors is that Nios II /e doesn't have pipeline and embedded multipliers, and Nios II /f has hardware improvements on execution time like dedicated hardware for functions and acceleration between pipeline stages. It's important to know that these results are valid for any soft-processor, and this specific choice has to do with researches experience with these tools and IDE's.

After initial experiments, the profile of the algorithm showed that the main problem is the fitness function followed by crossover function. To solve the initial problem, another six processors (Nios II /xf - FPU, Nios II /xfd - FPU+HW

Divisor) were configured and tested. Table II contains all information of the processors (Logic Elements(LE), Pins (P), Memory Bits (MB), Maximum Clock in MHz (Clock)) together with μGA execution time in seconds (ET) for each one.

TABLE II
COMPARING DEVELOPED HARDWARE

	ET	LE	MB	P	Clock
Nios II /e	0,40	3%	72%	$< 1\%$	108.80
Nios II /s	0,09	4%	75%	$< 1\%$	97.54
Nios II /f	0,07	5%	77%	$< 1\%$	113.06
Nios II /ef	0,29	4%	72%	$< 1\%$	106.26
Nios II /sf	0,07	6%	75%	$< 1\%$	89.66
Nios II /ff	0,06	7%	77%	$< 1\%$	116.21
Nios II /efd	0,29	11%	72%	$< 1\%$	107.70
Nios II /sfd	0,07	13%	75%	$< 1\%$	93.39
Nios II /ffd	0,06	14%	77%	$< 1\%$	113.34

Execution time was significantly reduced after including FPU unit without division hardware, but still the same including division hardware. In this case division hardware only increased the total area of the hardware and reduced processor clock performance.

TABLE III
CROSSOVER FUNCTION NUMBERS.

	calls	% of Total Time	Function Time
Nios II /ff	51	6,49	0,10ms
Nios II /ff + hw	51	0,00	0,03ms

The second problem is the crossover function. This function is called once for each generation and is a good function to have a hardware version. Using hardware implementation described in previous section as custom instruction of Nios II /ff soft-processor, the results are presented on table III. Developed hardware transformed one 'for' loop inside crossover routine into one macro call to the new hardware structure. For 51 calls of the function the time for crossover decreased from 0,10ms to 0,03ms. Final time only 30% of software time. For a high number of executions, this values are extremely significant with a 70% reduction on execution time of the function.

VII. CONCLUSIONS

The results presented on previous section show that the proposed architecture achieve the expected results meeting requirements of "soft" real-time hardware. Comparing to related works on section II, this work is a self-contained embedded system, because all data and information need for execution is implemented; the hardware is physically implemented on FPGA and μ can solve non-linear problems too [10]. Together with cited contributions, the proposed crossover with "shuffle" selection achieve good results for μGA.

This crossover hardware shows how simple is to implement "shuffle" selection with one cut-point crossover and Nios II custom instructions have another operand, and another templates. This fact allows implementations with larger individuals or large number of individuals with significative reduction on execution time keeping the need of few clock cycles. Moreover Nios II soft-processor allow 255 custom instructions to be add.

Future works tends to more complex fitness functions implementation and evaluation together with more hardware development, for the other operand (epidemic) for example, and real robot tests.

VIII. ACKNOWLEDGMENT

The authors acknowledge the support granted by CNPq and FAPESP to the INCT-SEC (National Institute of Science and Technology - Critical Embedded Systems - Brazil), processes 573963/2008-9 and 08/57870-9. Also acknowledge to CNPq for their financial support of this research (M.Sc. grant).

REFERENCES

[1] D. E. Goldberg, *Genetic Algorithms in Search, Optimization, and Machine Learning*, 1st ed. Addison-Wesley Professional, January 1989.

[2] A. L. Nelson, G. J. Barlow, and L. Doitsidis, "Fitness functions in evolutionary robotics: A survey and analysis," *Robot. Auton. Syst.*, vol. 57, no. 4, pp. 345–370, 2009.

[3] C. Bobda, *Introduction to Reconfigurable Computing: Architectures, Algorithms, and Applications*. Springer Publishing Company, Incorporated, 2007.

[4] P. Yiannacouras, J. Rose, and J. G. Steffan, "The microarchitecture of fpga-based soft processors," in *CASES '05: Proceedings of the 2005 int. conf. on Compilers, arch. and synthesis for emb. sys.* New York, NY, USA: ACM, 2005, pp. 202–212.

[5] Z. Zhu, D. J. Mulvaney, and V. A. Chouliaras, "Hardware implementation of a novel genetic algorithm," *Neurocomput.*, vol. 71, no. 1-3, pp. 95–106, 2007.

[6] N. Nedjah and L. de Macedo Mourelle, "Hardware architecture for genetic algorithms," in *IEA/AIE'2005*. London, UK: Springer-Verlag, 2005, pp. 554–556.

[7] P.-Y. Chen, R.-D. Chen, Y.-P. Chang, L.-S. Shieh, and H. Malki, "Hardware implementation for a genetic algorithm," in *Instrumentation and Measurement, IEEE Transactions on*, 2008, pp. 699–705.

[8] T. Oliveira and V. Pilla, "An implementation of compact genetic algorithm on fpga for extrinsic evolvable hardware," in *Programmable Logic, 2008 4th Southern Conference on*, 2008, pp. 187–190.

[9] S. Kher, T. Ganesh, P. Ramesh, and A. K. Somani, "Greedy dynamic crossover management in hardware accelerated genetic algorithm implementations using fpga," *Computer Modeling and Simulation, International Conference on*, vol. 0, pp. 47–52, 2009.

[10] K. KRISHNAKUMAR, "Micro-genetic algorithms for stationary and non-stationary function optimization," in *SPIE*, 1989, pp. 289–296.

[11] W. Wolf, "A decade of hardware/software codesign," *Computer*, vol. 36, no. 4, pp. 38–43, April 2003.

[12] S. T. Houssain, "An introduction to evolutionary computation," 1998.

[13] J. Straunstrup and W. Wolf, *Hardware Software Co-Design : Principles and Parctice*, 1st ed. Springer, 1997.

[14] G. de Micheli and R. K. Gupta, "Hardware/software co-design," in *Proceedings of IEEE, vol. 85*, 1997, pp. 349–365.

[15] M. Dias and W. Lacerda, "Hardware/software co-design using artificial neural network and evolutionaryy computing," in *5th SPL.*, April 2009, pp. 153–157.

[16] H. Hübert and B. Stabernack, "Profiling-based hardware/software co-exploration for the design of video coding architectures," *IEEE Trans. Cir. and Sys. for Video Technol.*, vol. 19, no. 11, pp. 1680–1691, 2009.

[17] K. E. E.-D. El-Sayed M. Saad, Medhat H. A. Awadalla, "Fpga-based software profiler for hardware/software co-design," in *26th NRSC2009*, 2009, pp. D14 1–8.

[18] P. Project, "The player project," http://playerstage.sourceforge.net/, 2010, acesso: 17/07/2010.

[19] A. Corp., "Altera.com," http://www.altera.com/literature, 2010, acesso: 22/05/2010.

[20] M. Honeyford, "Speed your code with the gnu profiler," http://www.ibm.com/developerworks/library/l-gnuprof.html, 2010, acesso: 17/07/2010.